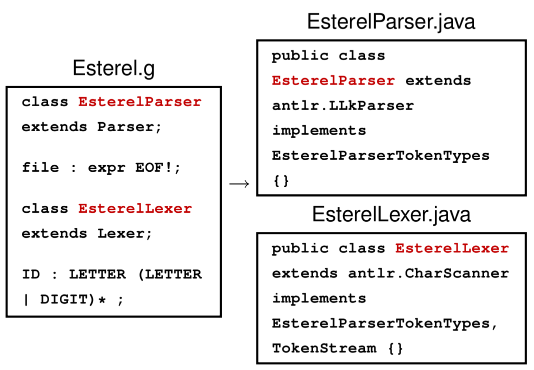
An ANTLR Grammar for Esterel

COMS W4115

Prof. Stephen A. Edwards
Fall 2005
Columbia University
Department of Computer Science

ANTLR


Esterel.g

```
class EsterelParser
extends Parser;

file : expr EOF!;

class EsterelLexer
extends Lexer;

ID : LETTER (LETTER
| DIGIT)* ;
```

→

EsterelParser.java

```
public class
EsterelParser extends
antlr.LLkParser
implements
EsterelParserTokenTypes
{}
```

EsterelLexer.java

```
public class EsterelLexer
extends antlr.CharScanner
implements
EsterelParserTokenTypes,
TokenStream {}
```

ANTLR Lexer Specifications

Look like

```
class MyLexer extends Lexer;
options {
  option = value
}

Token1 : 'char' 'char' ;
Token2 : 'char' 'char' ;
Token3 : 'char' ('char')? ;
```

Tries to match all non-protected tokens at once.

ANTLR Parser Specifications

Look like

```
class MyParser extends Parser;
options {
  option = value
}

rule1 : Token1 Token2
      | Token3 rule2 ;
rule2 : (Token1 Token2)* ;
rule3 : rule1 ;
```

Looks at the next k tokens when deciding which option to consider next.

An ANTLR grammar for Esterel

Esterel: Language out of France. Programs look like

```
module ABRO:
input A, B, R;
output O;

loop
  [ await A || await B ];
  emit O
each R

end module
```

The Esterel LRM

Lexical aspects are classical:

- Identifiers are sequences of letters, digits, and the underline character , starting with a letter.
- Integers are as in any language, e.g., `123`, and floating-point numerical constants are as in C++ and Java; the values `12.3`, `.123E2`, and `1.23E1` are constants of type double, while `12.3f`, `.123E2f`, and `1.23E1f` are constants of type float.
- Strings are written between double quotes, e.g., `"a string"`, with doubled double quotes as in `"a "" double quote"`.

The Esterel LRM

- Keywords are reserved and cannot be used as identifiers. Many constructs are bracketed, like "`present ... end present`". For such constructs, repeating the initial keyword is optional; one can also write "`present ... end`".
- Simple comments start with `%` and end at end-of-line. Multiple-line comments start with `%{` and end with `}%` .

A Lexer for Esterel

Operators from the langauge reference manual:

```
. # + - / * || < > , = ; : := ( )
[ ] ? ?? <= >= <> =>
```

Main observation: none longer than two characters. Need $k = 2$ to disambiguate, e.g., `?` and `??`.

```
class EsterelLexer extends Lexer;
options {
  k = 2;
}
```

A Lexer for Esterel

Next, I wrote a rule for each punctuation character:

```
PERIOD :        '.' ;
POUND :         '#' ;
PLUS :          '+' ;
DASH :          '-' ;
SLASH :         '/' ;
STAR :          '*' ;
PARALLEL :      "||" ;
```

A Lexer for Esterel

Identifiers are standard:

```
ID
  : ('a'..'z' | 'A'..'Z')
    ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*
  ;
```

A Lexer for Esterel

String constants must be contained on a single line and may contain double quotes, e.g.,

`"This is a constant with ""double quotes"""`

ANTLR makes this easy: annotating characters with ! discards them from the token text:

```
StringConstant
  : '"'!
    ( ~('"' | '\n')
    | ('"'! '"')
    )*
    '"'!
  ;
```

A Lexer for Esterel

I got in trouble with the ~ operator, which inverts a character class. Invert with respect to what?

Needed to change options:

```
options {
  k = 2;
  charVocabulary = '\3'..'\377';
  exportVocab = Esterel;
}
```

A Lexer for Esterel

Another problem: ANTLR scanners check each recognized token's text against keywords by default.

A string such as `"abort"` would scan as a keyword!

```
options {
  k = 2;
  charVocabulary = '\3'..'\377';
  exportVocab = Esterel;
  testLiterals = false;
}

ID options { testLiterals = true; }
  : ('a'..'z' | 'A'..'Z') /* ... */ ;
```

Numbers Defined

From the LRM:

Integers are as in any language, e.g., `123`, and floating-point numerical constants are as in C++ and Java; the values `12.3`, `.123E2`, and `1.23E1` are constants of type double, while `12.3f`, `.123E2f`, and `1.23E1f` are constants of type float.

Numbers

With $k = 2$, for each rule ANTLR generates a set of characters that can appear first and a set that can appear second. But it doesn't consider the possible combinations.

I split numbers into Number and FractionalNumber to avoid this problem: If the two rules were combined, the lookahead set for Number would include a period (e.g., from ".1") followed by end-of-token e.g., from "1" by itself).

Example numbers:	**First**	**Second**
`.1$`	.	EOT
`.2`	1	.
`1$`	2	1

Number Rules

```
Number
  : ('0'..'9')+
     ( '.' ('0'..'9')* (Exponent)?
       ( ('f'|'F') { $setType(FloatConst); }
       | /* empty */ { $setType(DoubleConst); }
       )
     | /* empty */ { $setType(Integer); }
     )
  ;
```

Number Rules Continued

```
FractionalNumber
  : '.' ('0'..'9')+ (Exponent)?
      ( ('f'|'F') { $setType(FloatConst); }
      | /* empty */ { $setType(DoubleConst); }
      )
  ;

protected
Exponent
  : ('e'|'E') ('+'|'-')? ('0'..'9')+
  ;
```

Comments

From the LRM:

Simple comments start with `%` and end at end-of-line.
Multiple-line comments start with `%{` and end with `}%`.

Comments

```
Comment
  : '%'
     ( ('{') => '{'
       (  // Prevent .* from eating the whole file
          options {greedy=false;}:
          (
            ('\r' '\n') => '\r' '\n' { newline(); }
            | '\r'                   { newline(); }
            | '\n'                   { newline(); }
            | ~( '\n' | '\r' )
          )
        )*
        "}%"
      | ((~'\n'))* '\n' { newline(); }
      )
  { $setType(Token.SKIP); }
  ;
```

A Parser for Esterel

Esterel's syntax started out using `;` as a separator and later allowed it to be a terminator.

The language reference manual doesn't agree with what the compiler accepts.

Grammar from the LRM

NonParallel:
  AtomicStatement
  Sequence

Sequence:
  SequenceWithoutTerminator `;` *opt*

SequenceWithoutTerminator:
  AtomicStatement `;` *AtomicStatement*
  SequenceWithoutTerminator `;` *AtomicStatement*

AtomicStatement:
  nothing
  pause
  ...

Grammar from the LRM

But in fact, the compiler accepts

```
module TestSemicolon1:
  nothing;
end module
module TestSemicolon2:
  nothing; nothing;
end module
module TestSemicolon3:
  nothing; nothing
end module
```

Rule seems to be "one or more statements separated by semicolons except for the last, which is optional."

Grammar for Statement Sequences

Obvious solution:

```
sequence
  : atomicStatement
    (SEMICOLON atomicStatement)*
    (SEMICOLON)?
  ;
```

```
warning: nondeterminism upon
k==1:SEMICOLON
between alt 1 and exit branch of block
```

Which option do you take when there's a semicolon?

Nondeterminism

```
sequence : atomicStatement
     (SEMICOLON atomicStatement)*
     (SEMICOLON)? ;
```

Is equivalent to

```
sequence : atomicStatement seq1 seq2 ;

seq1 : SEMICOLON atomicStatement seq1
     | /* nothing */ ;

seq2 : SEMICOLON
     | /* nothing */ ;
```

Nondeterminism

```
sequence : atomicStatement seq1 seq2 ;
seq1  : SEMICOLON atomicStatement seq1
      | /* nothing */ ;
seq2  : SEMICOLON
      | /* nothing */ ;
```

How does it choose an alternative in `seq1`?

First choice: next token is a semicolon.

Second choice: next token is one that may follow seq1.

But this may also be a semicolon!

Nondeterminsm

Solution: tell ANTLR to be greedy and prefer the iteration solution.

```
sequence
  : atomicStatement
    ( options { greedy=true; }
    : SEMICOLON! atomicStatement )*
    (SEMICOLON!)?
  ;
```

Nondeterminism

Delays can be "A" "X A" "immediate A" or "[A and B]."

```
delay : expr bSigExpr
      | bSigExpr
      | "immediate" bSigExpr ;

bSigExpr : ID
         | "[" signalExpression "]" ;

expr : ID | /* ... */ ;
```

Which choice when next token is an ID?

Nondeterminism

```
delay : expr bSigExpr
      | bSigExpr
      | "immediate" bSigExpr ;
```

What do we really want here?

If the delay is of the form “expr bSigExpr,” parse it that way.

Otherwise try the others.

Nondeterminism

```
delay : ( (expr bSigExpr) => delayPair
        | bSigExpr
        | "immediate" bSigExpr
        ) ;

delayPair : expr bSigExpr ;
```

The => operator means “try to parse this first. If it works, choose this alternative.”

Greedy Rules

The author of ANTLR writes

> I have yet to see a case when building a parser grammar where I did not want a subrule to match as much input as possible.

However, it is particularly useful in scanners:

```
COMMENT
  : "/*" (.)* "*/"
  ;
```

This doesn’t work like you’d expect...

Turning Off Greedy Rules

The right way is to disable greedy:

```
COMMENT
   : "/*"
     (options {greedy=false;} :.)*
     "*/"  ;
```

This only works if you have two characters of lookahead:

```
class L extends Lexer;
options {
  k=2;
}

CMT : "/*" (options {greedy=false;} :.)* "*/" ;
```

The Dangling Else Problem

```
class MyGram extends Parser;

stmt : "if" expr "then" stmt ("else" stmt)? ;
```

Gives

```
ANTLR Parser Generator   Version 2.7.1
gram.g:3: warning: nondeterminism upon
gram.g:3:      k==1:"else"
gram.g:3:      between alts 1 and 2 of block
```

Generated Code

```
stmt : "if" expr "then" stmt ("else" stmt)? ;
match(LITERAL_if);
expr();
match(LITERAL_then);
stmt();
if ((LA(1)==LITERAL_else)) {
 match(LITERAL_else);   /* Close binding else */
 stmt();
} else if ((LA(1)==LITERAL_else)) {
 /* go on: else can follow a stmt */
} else {
  throw new SyntaxError(LT(1));
}
```

Removing the Warning

```
class MyGram extends Parser;

stmt
  : "if" expr "then" stmt
    (options {greedy=true;} :"else" stmt)?
  ;
```

A Simpler Language

```
class MyGram
   extends Parser;

stmt
 : "if" expr
   "then" stmt
   ("else" stmt)?
   "fi"
 ;
```

```
match(LITERAL_if);
expr();
match(LITERAL_then);
stmt();
switch (LA(1)) {
case LITERAL_else:
  match(LITERAL_else);
  stmt();
  break;
case LITERAL_fi:
  break;
default:
  throw new SyntaxError(LT(1));
}
match(LITERAL_fi);
```